View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Waseda University Repository

2004年度修士論文

# 投機的スレッドを用いたキャッシュ効率化

## ～ハイパースレッディング環境を対象として～

提出日: 2005 年 2 月 2 日

指導: 山名早人助教授

早稲田大学大学院 理工学研究科 情報・ネットワーク専攻

学籍番号 ： 3603U125-3

本田　大

## 概 要

近年、CPU 処理速度と主記憶からのデータ転送速度との間の格差が顕著になってきているため、キャッシュメモリの重要性が高まってきている。よって、キャッシュメモリを有効活用するための最適化は、性能向上に欠かすことができず、盛んに研究されている。しかし、ポインタを利用した非線形なアクセスが多いプログラムや RDBMS などでは、キャッシュミスが頻発し、メモリアクセス遅延が隠蔽できない。キャッシュミスが頻発するような場合、キャッシュメモリによる速度向上は期待できないという問題がある。

この問題の解決策としてキャッシュにデータを事前にのせるための「Pre-Execution」が研究されている。しかし、いずれの Pre-Execution の提案においても、実行環境が専用ハードウェアを想定した、シミュレーションレベルでの実行に限定されており、実機上での投機的 Pre-Execution 手法の実用性は示されていない。また、従来の研究ではセマフォを使用した Producer-Consumer 方式によって、毎イタレーション Main Thread と Helper Thread 間で同期を取っている。このため、スレッド間の同期にかかるオーバヘッドが実行速度に大きく影響を与えるという問題がある。

こうした問題に対し、本論文では一般の CPU 上での投機的 Pre-Execution を実現するとともに、スレッド間での同期オーバヘッドの削減を目指す。具体的には、ハイパースレッディング環境における、投機的スレッドを利用したソフトウェアレベルでのキャッシュ効率化手法を提案する。また、静的に設定したイタレーション間の距離 (Thread Distance) によって、Helper Thread の待機・起動する手法を提案する。本手法によって、従来の毎イタレーション同期を取っていた手法に比べ、Main Thread と Helper Thread 間の同期回数を削減することができる。

Intel Xeon プロセッサでの実機上で、SPEC2000 INT 181.mcf、300.twolf、Olden ベンチマークの health において、スレッド間の同期をとる手法と非同期による手法で性能評価を行った。結果、既存のハードウェア構成を変えず、ソフトウェアによる並列化によって、平均 29.67%の 2 次キャッシュミスを削減し、181.mcf で 3.26%、health で 1.36%の処理性能高速化を達成できた。本実験による結果によって、従来の毎イタレーション同期を取る手法に比べ、27.1 %の性能向上を確認することができた。実機による性能向上はアーキテクチャの分野において、有益な結果である。

# 目 次

# 第1章　はじめに

近年、CPU 処理速度と主記憶からのデータ転送速度との間の格差が顕著になってきていることから、キャッシュメモリの重要性が高まってきている。キャッシュメモリとは、CPU 内部に設けられた高速な記憶装置のことである。キャッシュメモリに使用頻度の高いデータを蓄積しておくことにより、低速なメインメモリへのアクセスを減らすことができ、処理を高速化することができる。しかし、配列への非線形アクセスなどによって、キャッシュ・ミスが頻発するような場合、キャッシュによる速度向上は期待できない。

この問題の解決策として、プログラムで必要となる当該データを投機的に Load し、事前にキャッシュに載せることで、キャッシュ・ミスを低減する「Prefetch」や「Pre-Execution」が研究されている。

Prefetch とは、使用する可能性の高いデータが必要となる前にデータをメモリ階層の下層から上層へ取ってくる技術である。Prefetch にはソフトウェアによるアプローチ [9][10] と、ハードウェアによるアプローチ [11] がある。ソフトウェアによるアプローチでは、プログラマ、あるいはコンパイラがコード中に明示的なプリフェッチ命令を挿入し、ロードするアドレスを指定することで Prefetch を実現する。ハードウェアによるアプローチでは、動的にデータストリームを解析し、プリフェッチ命令を挿入することで Prefetch を実現する。

Pre-Execution とは、キャッシュミスを起こす可能性が高いロード命令および、分岐予測ミスを起こす可能性が高い分岐命令を、余剰な CPU 資源を利用して、単数あるいは複数の Helper Thread を実行することで、ロードすべきアドレスを計算し、Helper Thread による Prefetch をする技術である。Pre-Execution にもソフトウェアによるアプローチ [12] と、ハードウェアによるアプローチ [17] がある。ソフトウェアによるアプローチでは、プログラマ、あるいはコンパイラがプログラムコード中に、明示的な Pre-Execution 専用

の命令 (Helper thread 生成・待機・起動) を挿入することで Pre-execution を実現する。ハードウェアによるアプローチでは、データフローおよび、制御フローを実行時に解析し、動的に Helper thread が実行する命令群を決定し、Helper thread を実行することで Pre-Execution を実現する。しかし、いずれの提案においても、シミュレーションレベルでの実行に限定されており、実機上での投機的 Pre-Execution 手法は提案されていない。また、従来の研究ではセマフォを使用した Producer-Consumer 方式によって、毎イタレーション Main Thread と Helper Thread 間で同期を取っている。このため、スレッド間の同期にかかるオーバヘッドが実行速度に大きく影響を与えるという問題がある。

これらの問題に対し、本論文では一般の CPU 上での投機的 Pre-Execution を実現するとともに、スレッド間での同期オーバヘッドの削減を目指す。具体的には、ハイパースレッディング環境における、投機的スレッドを利用したソフトウェアレベルでのキャッシュ効率化手法を提案する。ハイパースレッディングとは、1 つの物理 CPU に 2 つの論理 CPU が搭載されているように見せる技術である。各論理 CPU がキャッシュを共有することから、１つの論理 CPU が使用したキャッシュメモリのデータを、もう一つの論理 CPU が参照可能である。本実験では、ハイパースレッディング環境を実装している Intel Xeon プロセッサ上において、提案手法によるキャッシュ効率化を検証した。必要とするデータを効率よくキャッシュに載せることで、キャッシュ・ミスによる性能低下を緩和し、全体の実行時間を短縮させることが目標である。また、Helper thread の起動と待機を、Main thread が実行したイタレーション数と Helper thread が実行したイタレーションの数の差を計算すること (Thread Distance) で決定し、従来の毎イタレーション同期をとる手法に比べ、同期回数を削減する手法を検証した。本手法によって、同期にかかるオーバヘッドを削減し、性能向上を見込むことができる。

本論文は全５章からなる。第２章では、キャッシュ機構の説明と関連研究について紹介し、第３章では、本研究の提案手法を説明する。第４章では本研究での実験結果を示し、考察を行う。第５章では、本研究での今後の課題を説明し、今後の方向を述べる。

# 第2章　関連研究

本章では、メモリ値の参照命令の性能への影響について説明する。次にキャッシュ機構と、キャッシュのヒット率について述べ、キャッシュ・ミスによる性能低下の軽減方法と、その関連研究について述べる。

## 2.1　メモリ値の参照命令の性能への影響

メモリ値の参照命令は、加算などの一般的な整数系演算命令に比べ、実行に時間がかかることが知られている。その原因としてデータ・キャッシュ・ミスによる遅延を挙げることができる。この遅延が生じる原因は、キャッシュへのアクセスよりもメインメモリへのアクセス速度が遅いからである。キャッシュ・ミスによるペナルティは大きく、性能に与える影響が大きい。ペナルティの大きさはメモリ・システムに依存する。

例えば、本論文で実験に用いる Intel Xeon プロセッサの記憶システムは L2 キャッシュまで備えている。Xeon の L1、L2 キャッシュ、メインメモリへのアクセスにかかるレイテンシを表 2.1 に示す。表 2.1 に示しているメモリアクセスレイテンシは Cache Burst 32[23] によって、計測した。

表 2.1: Xeon プロセッサにおけるメモリアクセスレイテンシ

| | |
|---|---|
| L1 キャッシュへのアクセスレイテンシ | 1clock |
| L2 キャッシュへのアクセスレイテンシ | 18clock |
| メインメモリへのアクセスレイテンシ | 213clock |

よって、メインメモリへのアクセスを減らし、必要なデータを L1、L2 キャッシュへとデータ供給するためのキャッシュ最適化が重要である。例えば、必要なデータを L2 キャッ

シュに載せることができれば、メインメモリアクセスに比較しアクセスレイテンシを約 $\frac{1}{20}$ にすることができる。

## 2.2 キャッシュメモリ機構

キャッシュメモリは、メモリウォール問題の解決を目的として開発された機構である。メモリウォール問題とは、プロセッサの高速化に対し、主記憶の高速化が追いつかず、プロセッサから見たメモリアクセスの速度が相対的に遅くなり、システムのパフォーマンスが低下するという問題である。

メモリアクセスには、一度参照されたデータは、近い将来にもう一度参照される可能性が高いという時間的局所性と、あるデータが参照されると、その近くにあるデータも、すぐに参照される可能性が高いという空間的局所性が存在する。キャッシュメモリは、これらの性質を利用して、プロセッサに近い場所に配置される、主記憶よりも高速な記憶装置のことである。頻繁に利用するデータをキャッシュメモリに格納しておくことで、プロセッサによる高速なメモリアクセスを可能にする。

主記憶の大容量化とプロセッサの高速化を両立させるために、主記憶の動作速度の何倍もの周波数で駆動するキャッシュメモリ機構は、近年の計算機アーキテクチャにおいて、必要不可欠な機構である。

## 2.3 キャッシュのヒット率

効率的なメモリアクセスを可能にするために、キャッシュのヒット率の向上を図る必要がある。プログラムの実行中にキャッシュ・ミスが起きる要因は、Compulsory,Capacity,Conflict の 3 つに分類することができる [1][2]。

1.Compulsory ミス

実行を開始してから、最初に必要になったブロックをアクセスする際に起きるミスである。初期ミス (initial miss) あるいは、コールドスタートミス (cold start miss) とも

呼ばれる。Compulsory ミスは、ラインサイズを大きくするか、prefetch をすることにより、ある程度軽減することができる。

2.Capacity ミス

キャッシュ容量が有限であるため、いったんキャッシュから主記憶に追い出したラインを、再びアクセスする際に起きるミスである。Capacity ミスは、キャッシュの容量を増やせば改善できるが、分岐予測機構やプロセッサダイ面積等等のトレードオフの関係にある。

3.Conflict ミス

セットアソシアティブマッピング方式キャッシュメモリにおいて、セットと主記憶アドレスとの対応関係が決まっているため、同一セットで競合したブロックが追い出され、再びアクセスされることにより生じるミスである。Conflict ミスは、セットアソシアティブの連想度を上げることによって改善できるが、実装上の制限がある。

## 2.4 データ・キャッシュ・ミスによる性能低下の軽減方法

データ・キャッシュ・ミスのペナルティによる性能低下を軽減するためには、次の 2 つの方法がある。

1. ペナルティが性能に与える影響を小さくする。
2. キャッシュ・ミス率を下げる。

1. のペナルティが性能に与える影響を小さくする技術として、次の手法がある。

(1)out-of-order 実行

(2)non-blocking cache の導入

out-of-order 実行とは、命令がフェッチされる順序と異なる順序で、命令を実行できる機能である。キャッシュ・ミスで実行が待ち合わせている間に、他の命令を out-of-order 実

行することで、キャッシュ・ミス・ペナルティが性能に与える影響を少なくすることができる。

non-blocking cache とは、キャッシュ・ミス時にも、他の命令のキャッシュ・アクセスに応答することができるキャッシュで、lock-up free cache[3][4] とも呼ばれる。non-blocking cache は資源競合を低減し、ILP(Instruction Level parallelism) により、ペナルティを隠蔽することができる。

out-of-order 実行と non-blocking cache を利用することで、キャッシュ・ミスに対する耐性を向上させることができる。しかし、キャッシュ・ミスを起こした命令が、クリティカル・パス上にある場合、キャッシュ・ミスのペナルティを隠蔽することができない。したがって、キャッシュ・ミスのペナルティ、あるいはキャッシュ・ミス率そのものを下げる技術が必要である。

2. のキャッシュ・ミスを下げる技術として、次の手法がある。

(1)Victim Cache

(2)Prefetch

(3)Pre-Execution

以下で、各々について詳細を説明する。

### 2.4.1 Victim Cache

(1) の Victim Cache は、キャッシュライン入れ換えの際に、追い出されるラインを一時的に蓄えておくバッファである [5]。図 2.1 に Victim Cache の概略図を示す。

図 2.1 において、L1 データキャッシュが追い出されたラインは、Victim Cache に一時的に保存される。その後、キャッシュ・アクセスがあった場合、CPU は L1 データキャッシュを参照するとともに、Victim Cache を参照する。L1 データキャッシュに目的のデータがなく、Victim Cache にデータがある場合、Victim Cache からデータを取り出す。取り出されたデータは、再び L1 キャッシュに保存される。Victim Cache は以上のように動作する機構である。

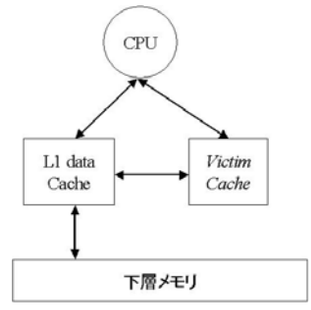


図 2.1: Victim Cache

Victim Cache は本来プログラム中に存在するメモリ参照の局所性を、キャッシュの容量によって利用できない場合を減らす技術である。

### 2.4.2 Prefetch

(2) の Prefetch とは、使用する可能性の高いデータが必要となる前に、データをメモリ階層の下層から上層へ取ってくる技術である。図 2.2 に Prefetch の概略図を示す。

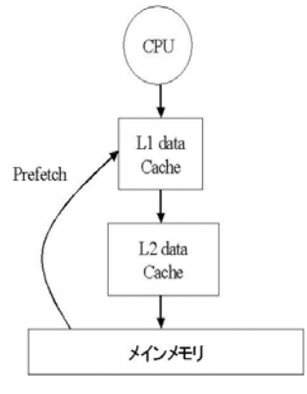


図 2.2: Prefetch

図 2.2 において、Prefetch を行うことにより、メインメモリからデータを L1 データキャッシュに持ってくることができる。Prefetch が成功すれば、真にデータが必要となったときに、キャッシュに必要なデータを供給することができるため、メインメモリへのアクセスによる遅延を防ぐことができる。Prefetch にはソフトウェアによるアプローチ [9][10] と、ハードウェアによるアプローチ [11] がある。

### 2.4.3 Pre-Execution

(3) の Pre-Execution とは、キャッシュミスを起こす可能性が高いロード命令および、分岐予測ミスを起こす可能性が高い分岐命令を、余剰な CPU 資源を利用して、単数あるいは複数の Helper Thread を実行することで、ロードすべきアドレスを計算し、Helper Thread による Prefetch をする技術である。図 2.3 に Pre-execution の概略図を示す。

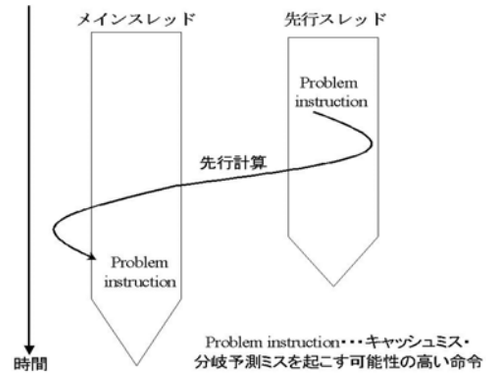


図 2.3: Pre-execution

図 2.3 において、Pre-execution とは、2つのスレッド（プログラムの実行のこと）を並行または並列に実行する手法である。Main thread はプログラム本体を実行し、Helper は上記プログラムのコピーまたは抜粋を実行する。また、Helper を Main thread よりも先に投機的に実行させる。Pre-execution は、従来の分岐予測や、Prefech でも分岐予測ミスやデータキャッシュミスが起こる可能性が高い命令 (図 2.3 における problem instruction) に対して適用される。つまり、Pre-execution とは、データ・プリフェッチ効果および、分岐予測や値予測の精度を向上させる技術である。Pre-Execution にはソフトウェアによるアプローチ [12] と、ハードウェアによるアプローチ [17] がある。

次節からは本研究に関連する、Prefetch および Pre-execution の研究を紹介する。

## 2.5 Software Prefetch

本節では、Prefetch のソフトウェアによるアプローチを説明する。Prefetch とは、使用する可能性の高いデータが必要となる前に、データをメモリ階層の下層から上層へ取って

くる技術である。ソフトウェアによるアプローチでは、プログラマ、あるいはコンパイラがコード中に明示的なプリフェッチ命令を挿入し、アドレスを指定することでPrefetchを実現する。Prefetchが成功すれば、真にデータが必要となったときに、キャッシュに必要なデータを供給することができるため、メインメモリへのアクセスによる遅延を防ぐことができる。

### 2.5.1 Greedy Prefetch[9]

1996年にカナダのToronto大学の、Todd C.Mowry、Chi-Keung Lukらが提案した、Greedy Prefetchingは、ポインタの移動先をすべてプリフェッチし、キャッシュに載せようとするprefetchの方法である。Greedy Prefetchingではコンパイラがデータフローを解析し、将来にデータが参照される前にコード中に明示的なプリフェッチ命令を挿入する。具体例として、リスト構造に適用する場合のコード例と概略図を図2.4に、２分木に適用する場合のコード例と概略図を図2.5に示す。

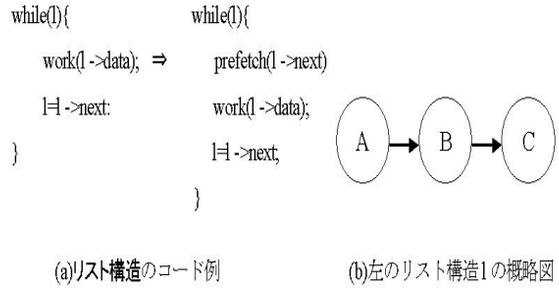


図 2.4: リスト構造に適用したとき [9]

図2.4の、リスト構造のデータをプリフェッチする場合の説明を行う。図2.4(b)のA、B、C、は図2.4(a)のコード例における、リスト構造の概略図を表している。Greedy Prefetchingは、リスト構造へのポインタがAであるときには、ポインタBをプリフェッチする。同様に、ポインタBである場合は、ポインタCをプリフェッチする。次に、図2.5の２分木のデータをプリフェッチする場合の説明を行う。図2.5(b)の1,2,3... は図2.5(a)のコード例の２分木tの概略図を表している。Greedy Prefetchingは、２分木へのポインタが１に位

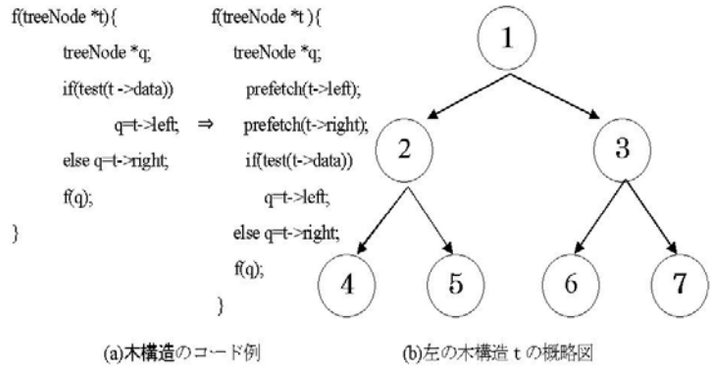


図 2.5: ２分木に適用したとき [9]

置していれば、データの２，３をプリフェッチする。その後、データ２にポインタが遷移したときには、データ４、５をプリフェッチする。Greedy Prefetchig は以上の動作によってプリフェッチを行う。

[9] の実験環境は MIPS R10000 のスーパスカラプロセッサに擬似したシミュレータである。コンパイラによって Prefetch 命令を挿入し、メモリアクセスレイテンシの軽減による性能向上を目的としている。Greedy Prefetch を適用することによって、Olden benchmark[1]において、-3%〜 31.4%の性能向上が見られたと報告している。

Greedy Prefetching における利点は、ポインタを移動させるためのオーバーヘッドが少ないこと、データのアクセス方法やデータ構造の更新を考慮せずに適用できること、およびコンパイラに直接実装させることができることである。

Greedy Prefetching における問題点は、静的なデータ解析によってのみプリフェッチを行うので、挿入したプリフェッチ命令の効果がない場合、動的に対処できないことである。

### 2.5.2 Prefetch Array[10]

2000 年にスウェーデンの Chalmers 大学の、Magnus Karlsson、Per Stenstron らが提案した Prefetch Array は、Greedy Prefetching[9] を拡張し、prefetch するデータの範囲を増やした手法である。Prefetch 命令を挿入する場合として、リスト構造に適用する場合の

[1] BH,Bisort,EM3D,Health,MST,Perimeter,Power,TreeAdd,TSP,Voronoi

コード例と概略図を図 2.6 に、２分木に適用する場合のコード例と概略図を図 2.7 に示す。

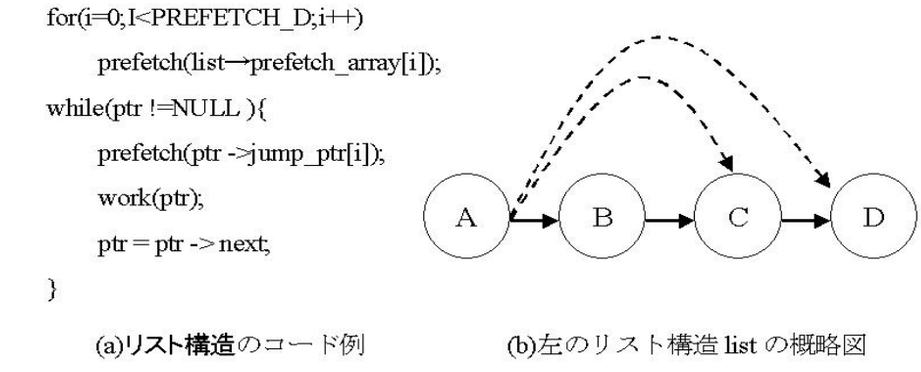


図 2.6: リスト構造に適用したとき [10]

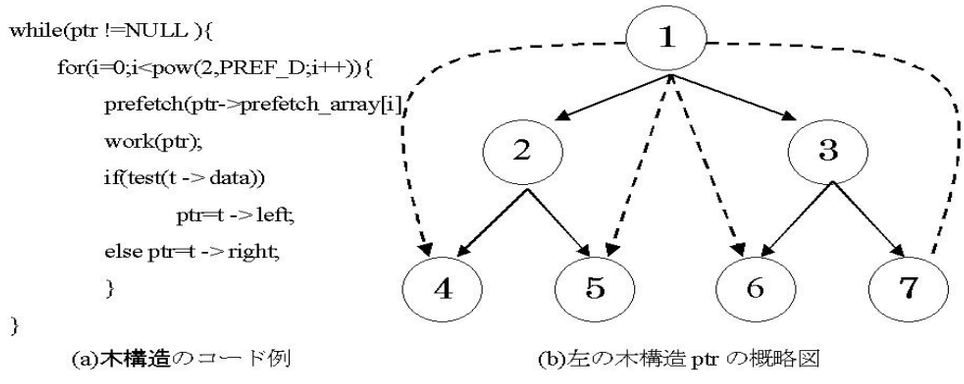


図 2.7: ２分木に適用したとき [10]

図 2.6 の、リスト構造のデータをプリフェッチする場合の説明を行う。図 2.6 において、(b) の A、B、C、D は (a) コード例のリスト構造 list の概略図を表している。Prefetch Array は、リスト構造へのポインタが A であるときには、データ B だけでなく、データ C、D もプリフェッチする。次に、図 2.7 の２分木のデータをプリフェッチする場合の説明を行う。図 2.7 において、(b) の 1,2,3... は (a) のコード例の２分木 ptr の概略図を表している。Prefetch Array は、２分木へのポインタが１に位置していれば、データの２，３をプリフェッチするだけでなく、データ４，５，６，７に対してもプリフェッチする。Prefetch Array は以上の動作によってプリフェッチを行う。

[10] の実験環境は SPARC v8 アーキテクチャをモデルとしたシミュレータである。[9] の

手法によって挿入されたPrefetch命令に対し、Prefetchするアドレスを増やすことによって、メモリアクセスレイテンシを軽減し、性能向上させることを目的としている。Prefetch Arrayを適用することで、Olden benchmark[2]において、20%〜40%の性能向上が見られたと報告している。

ただし、Prefetch Arrayの方式を実現するためには、greedy prefetching[9]に比べ、メインメモリへのアクセスによる遅延、Prefetch Array命令実行ためのオーバーヘッドやキャッシュの汚染を考慮しなければうまく機能しないという問題がある。よって、追加したPrefetch命令のアドレスの範囲と実行性能との関係を判断し、最も良いパフォーマンスを得るアルゴリズムを考案する必要がある。

## 2.6 Hardware Prefetch

本節では、Prefetchのハードウェアによるアプローチを説明する。Prefetchとは、使用する可能性の高いデータが必要となる前に、データをメモリ階層の下層から上層へ取ってくる技術である。ハードウェアによるアプローチでは、動的にデータストリームを解析し、プリフェッチ命令を挿入することでPrefetchを実現する。Prefetchが成功すれば、真にデータが必要となったときに、キャッシュに必要なデータを供給することができるため、メインメモリへのアクセスによる遅延を防ぐことができる。

### 2.6.1 Linked List Structure Prefetch[11]

1998年にWisconsin Madison大学のAmir Roth,Gurindar S.Sohiらが提案した、Dependence Based Prefetching [11]（以下DBP）は、構造体の連結リストへのアクセスに際し、将来アクセスする構造体へのアドレスを先行計算する方法である。図2.8にコード例とDBPの概略図を示す。

DBPは、プロセッサと別にPrefetch Engineというアドレス計算のためのハードウェア実行環境を備え、将来アクセスする構造体へのアドレスを先行計算した結果を保存するPrefetch Buffer(PB)を備える。図2.8において、連結リスト構造へのポインタfの命令が

[2]mst,mst.long,health DB.tree,treeadd,perimeter

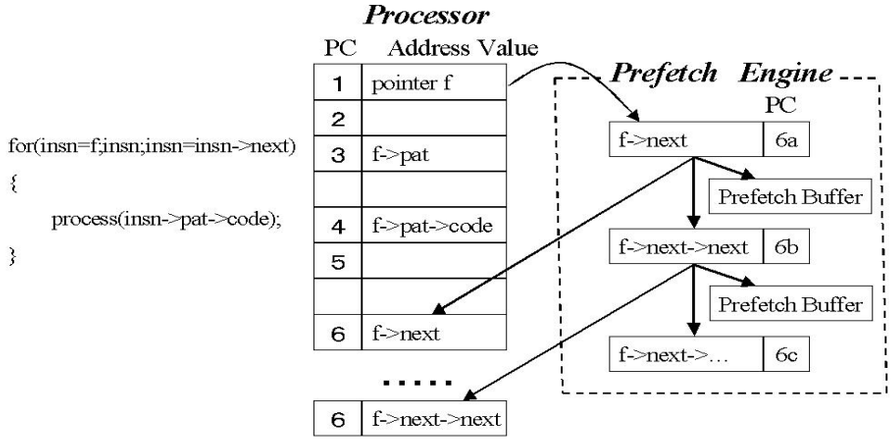


図 2.8: Dependence-based prefetching[11]

参照されたときに、Prefetch Engine は次の要素 (f->next) へのアドレスを計算する。この結果を PB に保存するとともに、さらに次の要素のアドレス (f->next->next) を計算する。プロセッサは通常のキャッシュと、PB の両方を参照することによって、キャッシュ・ミスを減らす。

ここで、Prefetch Engine を詳細に描いた図を図 2.9 に示す。

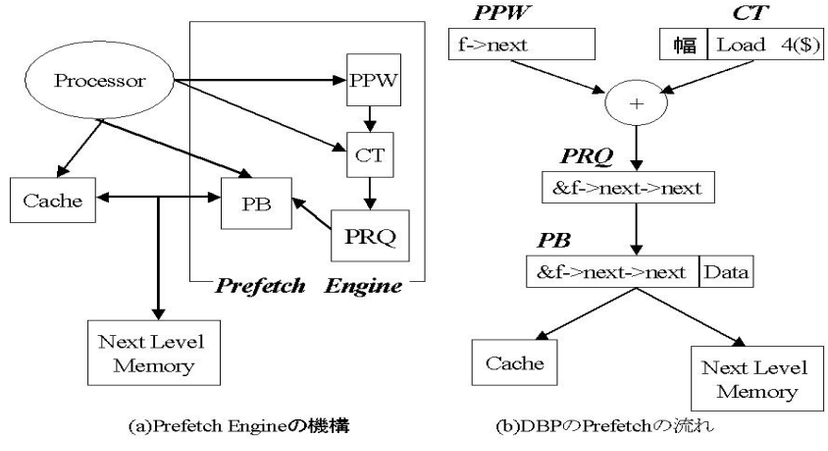


図 2.9: Prefetching Engine の詳細

図 2.9 の (a) では Prefetch Engine の具体的な機構を表し、(b) では DPG のプリフェッチの流れを表している。まず、(a) の Prefetch Engine の具体的な機構を説明する。Prefetch

EngineはPotential Producer Window(PPW)、Correlation Table(CT)、Prefetch Request Que(PRQ) を備える。PPW は、今までに参照した連結リスト構造のアドレスを保存する機構である。CT は、リスト構造のアドレス幅を保存する機構である。PRQ は、PPW と CT の和を計算した結果を保存し、そのアドレスを Prefetch Buffer(PB) に保存する機構である。PB は PRQ で計算されたアドレスおよび、そのアドレスのデータを保存する。

次に、(b) の DPG のプリフェッチの流れを説明する。連結リスト構造へのポインタが参照された場合、PPW には連結リスト構造のアドレス (f->next) が入っており、CT にはアドレス幅 (stride) が入っているので、これらの和をとったアドレスを PRQ に保存する。PRQ の結果判明した連結リストの次の要素 (f->next->next) へのアドレス、およびデータを PB に保存する。プロセッサは通常のキャッシュと、PB の両方を参照することによって、キャッシュ・ミスを減らす。

[11] では、スーパスカラプロセッサに、PPW、CT、PRQ、PB の Prefetch Engine を追加した実験環境でシミュレーションしている。DBP を適用することで、Olden benchmark[3] において、-2%〜 25%の性能向上が見られたと報告している。

DBP の問題点は、Prefech Buffer やキャッシュメモリに多数のアクセスポートを設けなければならないことである。つまり、多数のアクセスポートを設けることは、キャッシュと Prefetch Burffer へのアクセス制御を複雑にし、大規模にするという問題を引き起こしてしまう。

## 2.7 Software Pre-Execution

本節では、Pre-Execution のソフトウェアによるアプローチを説明する。ソフトウェアによるアプローチでは、プログラマ、あるいはコンパイラがプログラムコード中に、明示的な Pre-Execution 専用の命令 (Helper thread 生成・待機・起動) を挿入することで Pre-execution を実現する。

[3] BH,bis,em3,halth,perimeter,power,treeadd,tsp,voronoi

### 2.7.1 Speculative Pre-Computation[12]

California 大学の、Dean M.Tullsen,Intel の Joh P.Shen らが 2001 年に提案した、Speculative Precomputation（以下 SP）[12] はマルチスレッドプロセッサ上で、余剰のスレッドを使って、必要となるデータを早い段階で Prefetch する方法である。ハイパースレッディングテクノロジによって、シングル・スレッド・アプリケーションの、レイテンシを改善する。ハイパースレッディングとは、１つのプロセッサで複数のスレッドを処理し、プロセッサの実行資源を有効に利用して並列実行性を高め、性能の向上を図る技術のことである。SP の動作概要は次のとおりとなる。

1. キャッシュ・ミス・ペナルティによる、パフォーマンスの低下が大きいごく一部のロード命令 (以下 delinquent load) のみを、SP の実行対象とする。
2. それぞれの delinquent load ごとに、依存関係のある命令のみで構成される命令群 (以下 p-slice) を特定する。
3. ハードウェアに idle 状態のスレッドがあったときに、動的に p-slice を生成して、ロードアドレスの先行計算を投機的に行い、データを prefetch する。

以上の動作により、キャッシュ・ミスの大部分は命令実行でアドレス計算できるため、元のプログラムにおけるクリティカル・パスからメモリ・レイテンシを隠蔽することができる。つまり、SP とは、本来スレッドレベルの並列化 (Thread level Parallelism:TLP) のために用意されている実行資源を活用して、メモリ・レベルの並列化 (Memory level Parallelism:MLP) を高める技術である。

[12] では、Simultaneous Multithreading でのシミュレータによる実験で評価を行っている。また、スレッドごとにレジスタと L1 キャッシュを割り当て、L2 キャッシュやメインメモリは共有している状態で実験を行っている。SP を適用することによって、SPEC2000fp[4]、SPEC2000int[5]、Olden Benchmark[6]において、8 つのスレッドを用いたときに-5%〜169%の性能向上が見られたと報告している。

SP の問題点は、Helper Thread と Main Thread 間の同期は毎イタレーションとってい

[4] 179.art,183.equake,
[5] 164.gzip,181.mcf
[6] health,mst

るため、同期にかかるオーバヘッドが大きいことである。

## 2.8 Hardware Pre-Execution

本節では、Pre-Execution のハード ウェアによるアプローチを説明する。ハード ウェアによる Pre-Execution とは、データフローおよび、制御フローを実行時に解析し、動的に Helper thread が実行する命令群を決定し、Helper thread を実行することで Pre-Execution を実現する手法である。

### 2.8.1 分岐に対する Pre-Execution[17]

1998 年にフランスの Versailles 大学の Alexandre Farcy,Olivier Temam と、スペインの Catalunya 大学の Roger Espasa,Toni Juan らが提案した方法は、分岐の先行計算を行う方法である。条件式の内容が数式や、配列の線形的な参照による条件分岐判定である場合、アルゴリズムが単純であるので、真偽を判定しやすい。この具体例として、図 2.10 に具体的な分岐対象のコード例を示す。

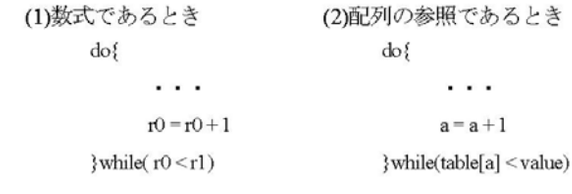


図 2.10: コード例

図 2.10 に、[17] において分岐命令の Pre-Execution 対象のコード例を示す。図 2.10 の (1) は、条件式の真偽を決める要因が数式である例である。ループの条件式を判定するための要因である r1 はループ Body では更新されない。したがって、分岐の真偽判定には r0 の計算のみを行うことで条件式の判定ができる。同様に、(2) では、条件式の真偽を決める要因が配列の参照である例である。ループの条件式を判定するための配列の要素の値および value の値はループ Body で更新されない。したがって、配列の線形的なアクセスを

行うことで分岐の真偽を判定することができる。この方式を適用した例 [17] を図 2.11 に示す。

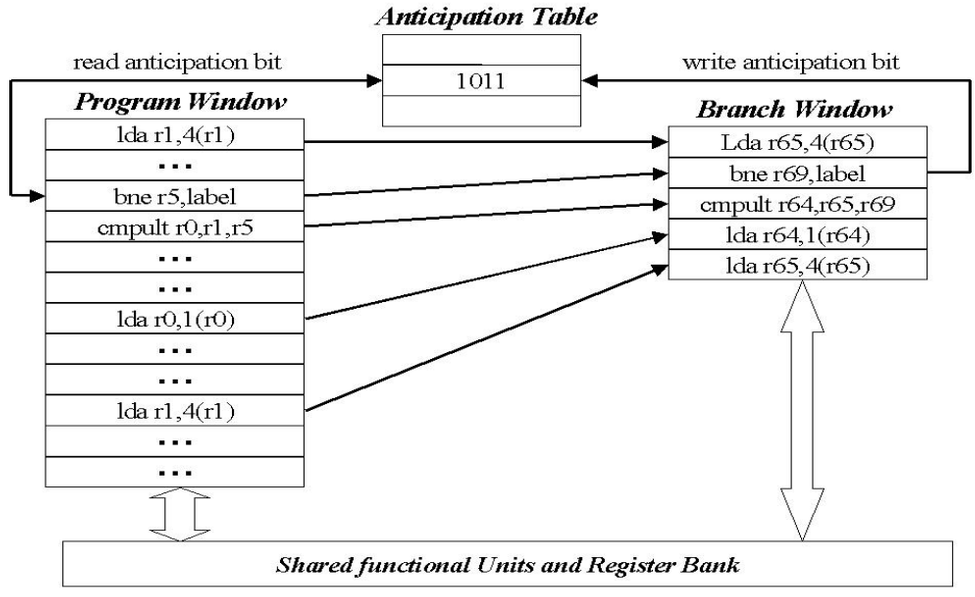


図 2.11: [17] の概略図

ここで、通常のプログラムの処理を program flow と定義し、通常のプログラムの処理から抽出した命令群 (図 2.11 の lda,bne,cmpult) を branch flow と定義する。また、program flow を実行する機構を Program Window と呼び、branch flow を実行する機構を Branch Window と呼ぶ。Program Window と Branch Window は同一のレジスタと Processor Unit(PU) を共有するが、Branch Window の方がレジスタや PU に優先してアクセスすることができる。また、Branch Window が先行計算した結果を保存しておく Anticipation Table を備える。よって、この手法は条件式を先行計算し、分岐先を確定することで条件分岐ミスを減らす手法である。

図 2.11 において、Program Window ではループ内のすべての命令を計算する。Branch Window では、ループの条件式に関する命令 (lda) および、分岐命令 (bne)、条件判定の命令 (cmpult) のみを計算し、分岐先のアドレスを Anticipation Table に保存する。そして、Program Window が Anticipation に保存されている先行計算された結果を参照することで、分岐予測ミスを減らすことができるのである。

[17] では、SPEC2000[7]において、対象とした条件式のみの分岐予測ミスのレイテンシを平均で 60%減らしたと報告している。

この手法の問題点は、図 2.10 のような条件式の分岐の決定要因が数式や、配列の参照のときのみである。したがって、非線形な分岐の決定要因の場合には適用しておらず、応用範囲が狭いという問題がある。

## 2.9 2 章のまとめ

本節では、第 2 章で紹介した関連研究のまとめを表 2.1 に示す

表 2.2: 第 2 章で紹介した関連研究のまとめ

| 手法 | アプローチ | 提案者 | アルゴリズム | 特徴 | 問題点 |
|---|---|---|---|---|---|
| Prefetch | Software | C-K.Luk (1996) | Greedy Prefetching[9] | ポインタの移動先のデータをすべて Prefetch する手法。アルゴリズムが簡単で実装しやすい。 | 静的なデータ解析によってのみ Prefetch を行い、挿入したプリフェッチ命令の効果がない場合、動的に対処できないことである。 |
| | | M.Karlsson (2000) | Prefetch array[10] | 連結リスト構造に対し、データのアドレス幅をもとに Prefetch を行う。 | Prefetch Array 命令実行ためのオーバーヘッドや、キャッシュの汚染を考慮する必要がある。 |
| | Hardware | A.Roth (1998) | DGP[11] | 連結リスト構造に対し、データのアドレス幅をもとに Prefetch を行う。 | Prefech Buffer やキャッシュメモリに多数のアクセスポートを設けなければならないことである。これによって、キャッシュと Prefetch Burffer へのアクセス制御を複雑にしてしまう。 |
| Pre-Execution | Software | J.Collins (2001) | SP[12] | SMT 環境における、Helper スレッドを用いた Prefetch 手法。 | Helper Thread と Main Thread 間の同期は毎イタレーションとっているため、同期に必要なオーバヘッドが大きい。 |
| | Hardware | Toni Juan (1998) | Dataflow Analysis[17] | ループの条件式に対して Pre-Execution する手法。 | 条件式の分岐の決定要因が数式や、配列の参照のときのみで、非線形な分岐の決定要因の場合には適用しておらず、応用範囲が狭いという問題がある。 |

[7] gcc,ijpeg,li,m88ksim,perl

第２章でのいずれの提案においても、実行環境が専用ハードウェアを想定したシミュレーションレベルでの実行に制限されており、実機上での投機的 Pre-Execution 手法の実用性は示されていない。また、[12] では、毎イタレーション Helper Thread と Main Thread 間で同期をとっているため、同期にかかるオーバヘッドが大きいという問題がある。

本論文では、ハイパースレッディング環境における、実機上での投機的スレッドを利用したソフトウェアレベルでのキャッシュ効率化手法の実用性を示す。また、静的に設定したイタレーション間の距離 (Thread Distance) によって、Helper Thread の待機・起動する手法を提案する。本手法を適用することによって、従来の毎イタレーション同期を取っていた手法に比べ、Main Thread と Helper Thread 間の同期回数を削減し、性能向上を見込むことができる。

# 第3章　提案手法

本章では、ハイパースレッディング環境における、投機的スレッドを利用したソフトウェアレベルでのキャッシュ効率化手法を提案する。また、静的に設定したイタレーション間の距離 (Thread Distance) によって、Helper Thread の待機・起動する手法を提案する。本手法を適用することによって、従来の毎イタレーション同期を取っていた手法に比べ、Main Thread と Helper Thread 間の同期回数を削減し、性能向上を見込むことができる。

## 3.1　対象アーキテクチャ

本提案手法は、オンチップマルチプロセッサで、複数の CPU がキャッシュを共有しているアーキテクチャを対象とする。本論文では、その中でも特に Intel の CPU を対象とした。Intel の SMT(Simultaneous Multi-Threading) アーキテクチャ対応プロセッサは、Hyper-Threadingを実装している。Hyper-Threadingに対応した CPUでは、メインスレッドと Helper スレッド間で、L1 キャッシュ、L2 キャッシュ等の実行資源を共有できる。このように、論理 CPU が 2 つあると仮定できるので、1 本の Main Thread と、1 本の Helper Thread、つまり合計 2 本のスレッドによる実行を仮定した。

## 3.2　プログラムのモデル化

Hyper-Threading 技術を利用し、Helper Thread を Main Thread に先行して実行させる。Helper Thread は Main Thread の処理中に、後で必要となるデータを prefetch することによって、Main Thread のキャッシュミスに起因するストールを隠蔽する。本手法のモデル図を図 3.1 に示す。プログラムの流れどおりの処理をするスレッドを「Main thread」と定義し、プログラムの一部の関数を処理するスレッドを「Helper thread」と定義する。

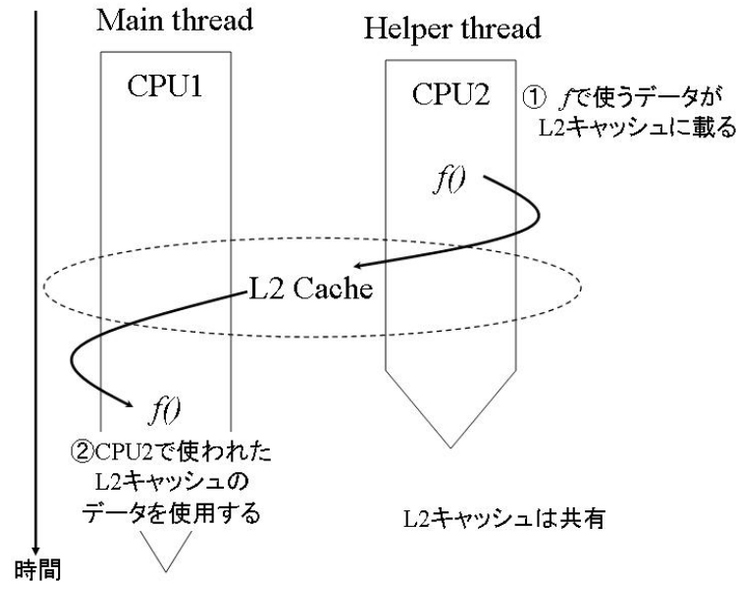


図 3.1: 実行の流れ

Main thread に対し、Pre-Execution 用の Helper thread を 1 本作成する。Helper thread はデータをキャッシュに載せるためだけに使用する。そのため、Helper thread では、メインスレッドに影響を与えるストア命令を発行しない。これにより、Main thread における実行の正確性を保証し、かつ、Helper thread における命令数を削減できる。

本手法の実行モデルによる実行の流れを以下に記す。

1. CPU1 に Main thread を割り当てる。Main thread が実行するのは図 3.1 における *f()* を実行する。
2. CPU2 に Helper thread を割り当てる。Helper thread は Main Thread が将来実行するコード（図 3.1 における *f()*）を実行する。

この実行モデルによって、L2 キャッシュは下記のとおりに機能すると考えられる。

(1) *f()* で使うデータが L2 キャッシュに載る。

(2) CPU1 が、CPU 2 で使われた L2 キャッシュのデータを参照する。

(3) アクセスレイテンシの軽減による速度向上が見込める。

## 3.3 プログラムの実行フロー

本手法では、Pre-Execution の適用対象をプログラムの最内ループとする。Main thread が Pre-Execution の適用対象ループに入るまでは、シングルスレッドのみで実行する。Pre-Execution 対象ループに入った直後、Helper thread を作成し、2 スレッドによる並列処理を開始する。アプリケーションの動作を図 3.2 に示す。

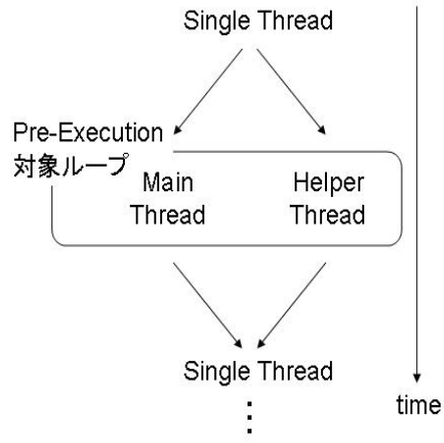


図 3.2: アプリケーションの実行フロー

## 3.4 Helper thread の生成

Helper thread の作成には、WIN32API の CreateThread() を用いて実装した。Helper thread が実行する命令は、delinquent ロードと依存関係がある命令のみで構成する。

## 3.5 スレッド間の同期モデル

Helper thread の実行が Main thread の実行より、先行しすぎないために、スレッド間で同期をとる必要がある。なぜなら、Main thread が必要とするデータを使用する前に、Helper thread によって、prefetch しなければならないからである。本手法の同期モデルを図 3.3 に示す。

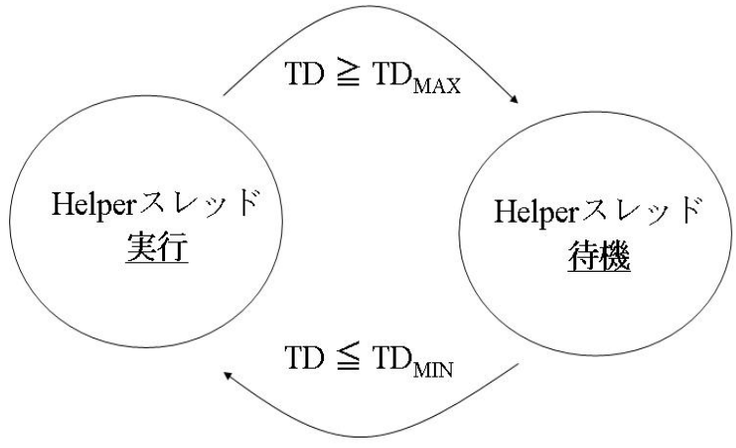


図 3.3: スレッド間の同期手法

図 3.3 における Thread Distance（以下 TD）とは、Helper thread と Main thread とのイタレーションの距離である。ここでいうイタレーションの距離とは、同一ループ内における Main thread が実行中のイタレーションより、Helper thread が何イタレーション先行しているかを示す。$TD_{MAX}$ とは、Helper thread が Main thread に比較して先行しすぎないようにするために静的に設定した TD の値である。また、$TD_{MIN}$ とは Helper thread を待機状態から復帰させるために静的に設定した値である。つまり、Helper thread は、TD が $TD_{MAX}$ 以上になった際に実行を停止し、待機状態に入る。その後、待機状態で TD を計算し、TD が $TD_{MIN}$ 以下になった際に再び Helper thread の実行を開始させる。2 次キャッシュに載る最適な $TD_{MAX}$ で実装することで、Pre-Execution 適用範囲を拡大し、prefetch の量を増加による、速度向上が見込める。

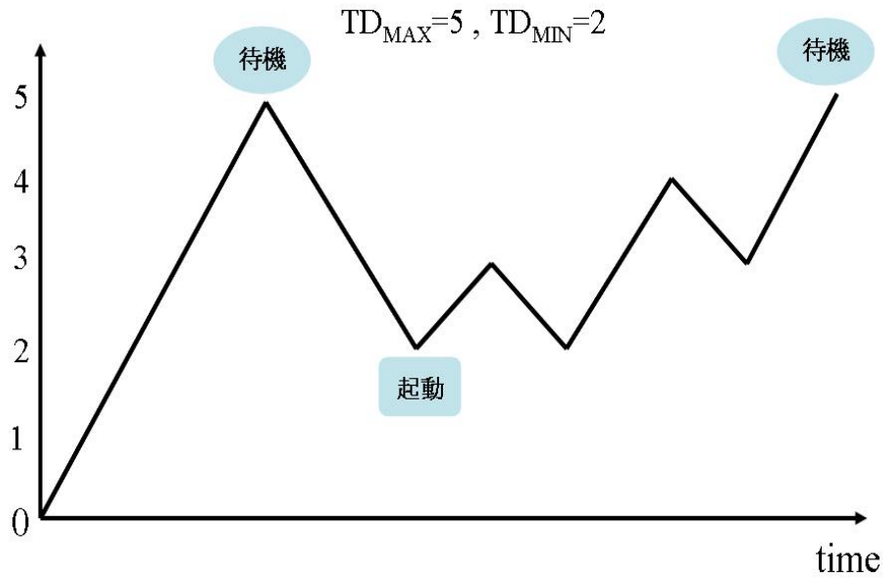


図 3.4: 本提案手法における Helper スレッドの待機と起動

図 3.4 に $TD_{MAX}$=5 と $TD_{MIN}$=2 の場合での、本提案手法における Helper thread の待機と起動させるタイミングを示す。最初は TD が 5 になるまで、Helper thread と Main thread は並列に実行し続ける。TD が $TD_{MAX}$ である 5 以上になった場合、Helper thread は Pre-Execution しすぎないようにするために待機する。Helper thread が次に起動するのは、Main thread が Pre-Execution 対象ループのイタレーションを実行し続けた後に TD の値が $TD_{MIN}$ である 2 以下になった場合である。後は、TD が 5 になるまで、Helper thread と Main thread を並列に実行させ続ける。この手法によって、従来の毎イタレーション同期を取る手法に比べ、同期オーバヘッド、すなわち待機と起動の回数を削減することができ、速度向上が見込める。ただし、$TD_{MAX}$ の値を大きくしすぎると、L2 キャッシュに prefetch したデータが収まらなくなるために、逆に性能低下が発生すると考えられる。

本手法では、同期をとるために Main thread での実行済みイタレーション数を、グローバル変数としてインプリメントした。Main thread は、グローバル変数に現在まで実行したイタレーションの数を書き込む。このグローバル変数の値と Helper スレッドの実行済みイタレーション数の差が、静的に設定した値 $TD_{MAX}$ 以上になると、Helper thread は Pre-Execution を停止し待機する。次に、Helper thread は、グローバル変数と Helper thread が実行済みであるイタレーション数の差が、静的に設定した $TD_{MIN}$ の値以下になると、待機状態から復帰する。この手法が有効である理由は、Helper thread の命令数が Main thread に比べ短く、Helper thread が常に先行して実行できるためである。なお、スレッドの待機はスピンロックで実装している。

# 第4章　実験結果

本章では、第 3 章で説明した手法の有効性を検証するための評価環境と実験結果を示す。アプリケーションをシングルスレッドで実行した場合、2 次キャッシュミスが頻発するアプリケーションを評価対象とした。この評価対象アプリケーションに対し、シングルスレッドで実行した場合と、本手法を適用した場合の 2 次キャッシュミス数を計測し評価を行った。また、対象アプリケーションをシングルスレッドで実行した場合と本手法を適用した場合とのプログラム全体の実行時間を求め、速度向上率を算出することによって評価を行った。

## 4.1 評価環境

実行環境を表 4.1 に示す。2 次キャッシュミスを測定するツールとして、VTune Performance Analyzer 7.1 を使用した。実行時間の計測には、timeGetTime() 関数を使用した。

表 4.1: 実行環境

| | |
|---|---|
| CPU | Intel Xeon 2.4GHz |
| L1 Data Cache | 8KB,4-way-set associative 32 byte Line |
| L2 Data Cache | 512KB,8-way-set associative 64 byte Line |
| Main Memory | 1GB |
| OS | Windows XP Professional SP2 |
| Compiler | Intel C++ Compiler 8.0.48 |
| Compile Option | /Zi /Zd /QaxN /O3 |
| Link Library | winmm.lib |

実行環境における、キャッシュミスにおけるアクセスレイテンシを表 4.2 に示す。この評価には、Cache Burst[8] を用い、10 回の平均をとることで求めた。

表 4.2: キャッシュミスレイテンシ

| | |
|---|---|
| L1 Cache Access | 1 Cycle |
| L1 Cache Access | 18 Cycle |
| Main Memory Access | 213 Cycle |

L2 キャッシュへのアクセスレイテンシと比較して、メインメモリへのアクセスは、レイテンシが約 11.8 倍となっている。そこで、メインメモリへのアクセスを減らすことで計算機の速度向上を図る。

## 4.2 評価対象

表 4.3 に本手法で評価したプログラムを示す。

表 4.3: 実験対象アプリケーション

| ベンチマーク | アプリケーション | 入力セット |
|---|---|---|
| SPEC2000 INT | 181.mcf | ref |
| | 300.twolf | ref |
| Olden | health | 5 Level 2,048 Node |

今回の実験の評価プログラムとして、SPEC2000 から、181.mcf、300.twolf を選択した。また、Olden ベンチマークから health を選択した。VTune による解析結果から、これらのプログラムは、シングルスレッドで実行した際に、2 次キャッシュミスが多く発生するアプリケーションであることが判っている。2 次キャッシュミス数の大部分は、一部の命令に起因している。181.mcf の場合、キャッシュミスが頻繁に発生する命令は、双方向リストにおけるポインタの参照先アドレスを対象としたロード命令である。181.mcf における、delinquent ロードの例を図 4.1 に示す。図 4.1 では、SPEC2000 181.mcf での最も 2 次キャッシュミス数が多い関数を例としてとりあげた。

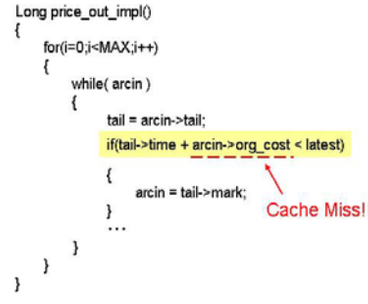


図 4.1: delinquent ロード位置

## 4.3 評価結果

本節では、第 4 章第 1 節で示した評価方法でプログラムを実行し、本提案手法での実験結果と考察を述べる。本実験では、Pre-Execution の実装は、2 次キャッシュミス数が最も多い関数中の最内ループに対して適用した。本実験で最も効果が得られた $TD_{MAX}$ と $TD_{MIN}$ の組み合わせ時における 2 次キャッシュミス数と削減率を表 4.4 に示す。表 4.4 に示すように、すべての関数において、2 次キャッシュミス数を削減できている。なお、$TD_{MAX}$ と $TD_{MIN}$ が 0 のときは、スレッド間で同期をしない、非同期実行を意味している。

表 4.4: 本手法の実験結果

| ベンチマーク | 関数名 | TD MAX | TD MIN | 提案手法適用前の 2 次キャッシュミス数 | 提案手法適用後の 2 次キャッシュミス数 | 2 次キャッシュミス削減数 |
|---|---|---|---|---|---|---|
| SPEC2000 | price_out_impl() | | | 2,230,461,558 | 1,497,031,268 | 32.88% |
| 181.mcf | compute_red_cost() | 4 | 3 | 268,263,348 | 162,418,341 | 39.46% |
| 300.twolf | new_box_a() | 3 | 2 | 1,114,842,354 | 758,598,612 | 31.95% |
| Olden health | check_patient_waiting() | 0 | 0 | 2,102,17,440 | 1,925,713,844 | 8.39% |

### 4.3.1 SPEC2000 INT 181.mcf

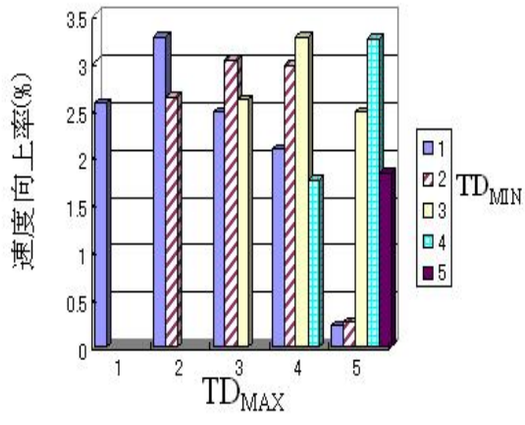


図 4.2: 速度向上率

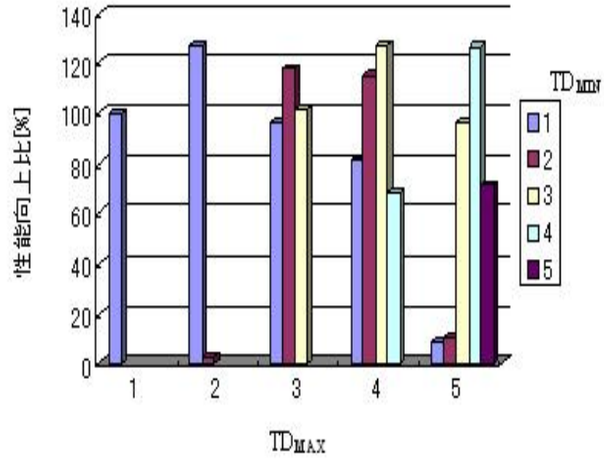


図 4.3: 性能向上率

図 4.2、図 4.3 に S PEC2000 181.mcf に本手法を適用した結果を示す。図 4.2 は Pre-Execution を適用しない場合を基準としたときの、本手法を適用した場合の速度向上率を示している。図 4.3 は従来の毎イタレーション同期をとる手法を基準にしたときの本手法を適用した場合の性能向上率を示している。

図 4.2 に示すように、$TD_{MAX}$=4、$TD_{MIN}$=3 のとき、最も高い速度向上率が得られた。また、図 4.3 に示すように、$TD_{MAX}$ と $TD_{MIN}$ との差が適切な場合、従来手法に比べ、性能が向上しているのがわかる。

181.mcf では、Helper thread の方が Main thread に比べ、実行コードが短いので、図 4.2 に示すように、提案手法が有効に機能していることがわかる。$TD_{MAX}$ と $TD_{MIN}$ の差が大きい場合、Helper thread が prefetch したデータが 2 次キャッシュからはずれ、Pre-Execution の効果が低減したと考えられる。

### 4.3.2 SPEC2000 INT 300.twolf

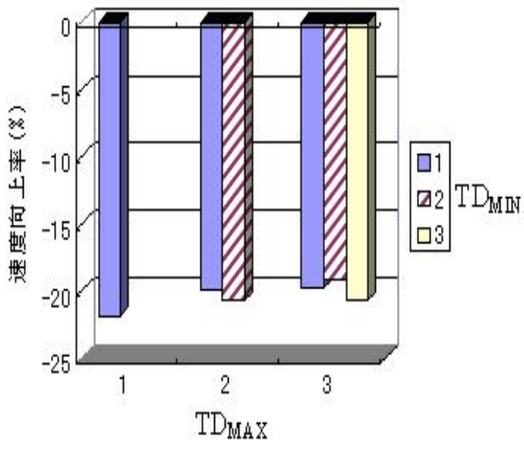


図 4.4: 速度向上率

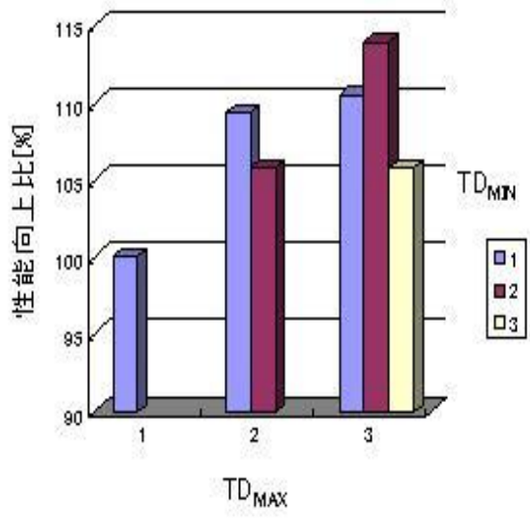


図 4.5: 性能向上率

図 4.4、図 4.5 に SPEC2000 300.twolf に本手法を適用した結果を、図 4.6、図 4.7 に Olden health に本手法を適用した結果を示す。図 4.4, 図 4.6 は Pre-Execution を適用しない場合を基準としたときの、本手法を適用した場合の速度向上率を示している。図 4.5, 図 4.7 は従来の毎イタレーション同期をとる手法を基準にしたときの本手法を適用した場合の性能向上率を示している。

しかし、300.twolf や health に本手法を適用した場合、約 15 〜 19%速度が低下した。図 4.5, 図 4.7 に示すように、毎イタレーション同期をとる手法より、性能が向上しているのがわかる。

300.twolf では、Pre-Execution 対象関数での最内ループのイタレーション回数が数回しかなく、スレッドの起動にかかるオーバヘッドが大きかったため速度低下が生じたと考えられる。また、health で速度低下が生じた原因として、Pre-Execution 対象関数での Main thread の実行コード数が非常に少ないため、Helper thread の起動・停止にかかるオーバヘッドが相対的に大きくなったためであると考えられる。

### 4.3.3 Olden health

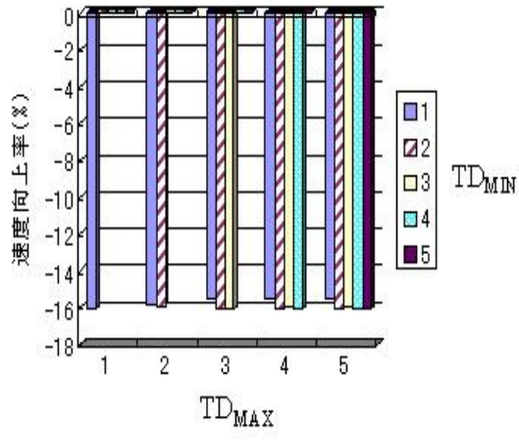


図 4.6: 速度向上率

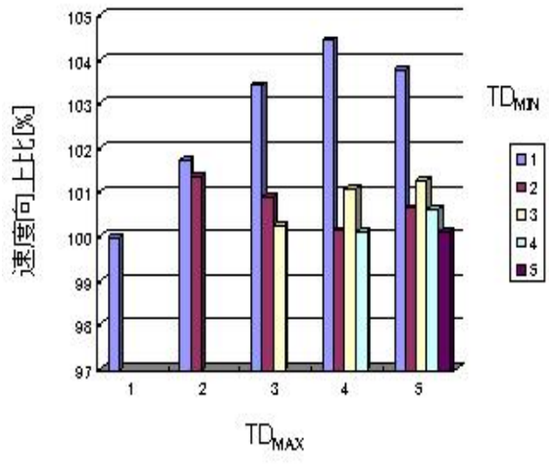


図 4.7: 性能向上率

つまり、Helper thread が Main thread に追いつかれたため Pre-Execution の効果が得られず、同期にかかるオーバヘッド分だけ速度低下が生じたと考えられる。よって、Olden health において非同期で Pre-Execution を実行したところ、本手法を適用しない場合と比較して、1.36%の速度向上率が得られた。ここで、非同期での実行とは、$TD_{MAX}$=0、$TD_{MIN}$=0 で実行することである。

# 第5章　おわりに

近年、CPU 処理速度と主記憶からのデータ転送速度との間の格差が顕著になってきていることから、キャッシュの重要性がより高まってきている。しかし、配列への非線形アクセスなどによって、キャッシュミスが頻発するような場合、キャッシュによる速度向上は期待できない。

本論文では、ハイパースレッディング環境における、投機的スレッドを利用したソフトウェアレベルでのキャッシュ効率化手法を提案した。また、Main thread と Helper thread 間の同期手法に関する提案と実装を行い、Pre-Execution 適用範囲を拡大した Pre-Execution 手法を Intel Xeon プロセッサ上で検証した。結果、2 次キャッシュミス数を平均 29.67%削減し、実行時間を最大 3.26%短縮することができた。

今後も CPU 速度とメモリ速度の差を埋めるための、キャッシュミスを減らす研究は性能向上のために必要不可欠である。また、スレッド間のオーバヘッドを軽減するためにも、新しい同期法の考案が必要だと考える。特に、同期あるいは非同期に Helper thread を起動させるかを自動的に判別するようなコンパイラが必要とされる。さらに、現在の Hyper Threading 技術を拡張し、論理 CPU を増やすことで、割り当てる Helper thread の数も CPU 台数に対応させ、Pre-Execution の効果をより高めていきたい。

# 謝辞

本研究の一部は，21 世紀 COE プログラム「プロダクティブ ICT アカデミア」によるものである．

本研究にあたりたくさんの指導、助言を頂いた早稲田大学理工学部情報学科山名早人助教授に感謝の意を表します。

# 参考文献


[1] 馬場　敬信:コンピュータアーキテクチャ, 第３章４節、オーム社、pp.166-217 (2000)

[2] David A.Patterson,John L.Hennessy:コンピュータの構成と設計　下巻　７章, 日経 BP 社,pp500-590 (1999)

[3] D.Kroft:Lockup-Free Instruction Fetch/Prefetch Cache Organization,In Proc.of 8th International Symposium on Computer Architecure(ISCA),pp.81-87,May 1981

[4] K.I.Farkas and N.P.Jouppi:Complexity/Perfomance Tradeoffs with Non-Blocking Loads,In Proc. of 21th International Symposium on Computer Architecure(ISCA),pp.211-221,April 1994

[5] N.P.Jouppi:Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,In Proc. of 17th International Symposium on Computer Architecure(ISCA),pp.364-373,May 1987

[6] M.D.Hill:Aspect of Cache Memory and Instruction Buffer Performance,Ph.D.thesis,Technical Report UCB/CSD87/381,University of California at Berkeley,Computer Science Division,November 1987

[7] Manoj Franklin and Gurindar S.Sohi:A Hardware Mechanism for Dynamic Reordering of Memory References,IEEE Transactions on Computers,Vol45,No.5,pp.552-571,May 1996

[8] Srindhar Gopal,T.N.Vijaykumar,James E.Smith,and Gurindar S.Sohi:Speclative Versioning Cache,In Proc.of 4th International Sympsium on High Performance Computer Architecture(HPCA),pp.195-205 (1998)

[9] C-K.Luk,Todd.C.Mowry:Compiler-based Prefetching for Recursive Data Structures,In Proc.of International Conference on Architectual Support for Programming Languages and Operating System(ASPLOS),pp.222-233 (1996)

[10] M.Karlsson,F.Dahlgren,P.Stenstrom:A Prefetching Technique for Irrregular Accesses to Linked Data Structures,In Proc. of 6th International Sympsium on High Performance Computer Architecture(HPCA),pp.206-217 (2000)

[11] A.Roth,A.Moshovos,G.Sohi:Dependence based prefetching for liked data structures,In Proc.of 8th International Conference on Architectual Support for Programming Languages and Operating System(ASPLOS),pp.115-126 (1998)

[12] J.Collins,H.Wang,D.Tullsen,C.Hughes,Y-F.Lee,D.Lavery,J.Shen:Speculative Precomputation:Long-range Prefetching of Delinquent Loads,In Proc.of 28th International Symposium on Computer Architecture(ISCA),pp.14-25 (2001)

[13] R.Chappell,J.Stark,S.Kim,S.Renhardt,Y.Patt:Simltaneous Subordinate Microthreading(SSMT),In Proc.of 26th International Symposium on Computer Architecture(ISCA),pp.186-195 (1999)

[14] C. K. Luk:Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors,In Proc. of 28th International Symposium on Computer Architecture(ISCA),pp.40-51 (2001)

[15] ]Dongkeun Kim,Donald Yeung:Design and Evaluation of Compiler Algorithms for Pre-Execution,In Proc. of 9th Intl. Conference on Architectual Support for Programming Languages and Operating System(ASPLOS) ,pp.62-73,(2002)

[16] C.Zilles,G.Sohi:Execution-based prediction using speculative slices,In Proc.of 28th International Symposium on Computer Architecture(ISCA),pp.2-13 (2001)

[17] Alexandre Farcy,Okiver Teman Roger Espasa,Toni Juan :Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes,In Proc.of 31st International Symposium on Microarchitecture(MICRO),pp.59-68 (1998)

[18] M.Annavaram,J.Patel,E.Davidson:Data Prefetching by Dependence Graph Precomputation,In Proc.of 28th International Symposium on Computer Architecture(ISCA),pp.52-61 (2001)

[19] K.Sundaramooorthy,Z.Purser,E.Rotenberg:Slipstraem Processor:Improving both Performance and Fault Tolerance,In Proc. of 9th International Conference on Architectual Support for Programming Languages and Operating System(ASPLOS),pp.257-268 (2000)

[20] Lance Hammon,Ben Hubbert,Michael Siu,Manohar Prabhu,Mike Chen,and Kunle Oluktun:The Stanford Hydra Chip MultiProcessor,IEEE MICRO Magagine,pp.71-84, March (2000)

[21] S.Wallace,B.Calder,and D.M.Tullsen:Threaded Multiple Path Execution,In Proc. of 25th International Symposium on Computer Architecture (ISCA),pp.238-249,June (1998)

[22] 村上　和希, 弘中　哲夫, 吉田　典可:投機的データプリフェッチを行うキャッシュの一考察, 情処研報（ARC),Vol.2000,No.140,pp.31-36 (2000)

[23] Cache Burst 32
http://user.rol.ru/ dxover/cburst/

[24] 本田 大, 斎藤 史子, 山名 早人:ハイパースレッディング環境における投機的スレッドを用いたキャッシュ効率化, 情報処理学会研究報告 2004-SWoPP-8,pp.43-48

[25] 本田 大, 斎藤 史子, 山名 早人:ハイパースレッディング環境における投機的スレッド間の同期手法の提案, 情報処理学会研究報告 2005-SHINING-8,pp.33-38

# 研究業績

[1] 本田 大, 斎藤 史子, 山名 早人:ハイパースレッディング環境における投機的スレッドを用いたキャッシュ効率化, 情報処理学会研究報告 2004-SWoPP-8,pp.43-48

[2] 本田 大, 斎藤 史子, 山名 早人:ハイパースレッディング環境における投機的スレッド間の同期手法の提案, 情報処理学会研究報告 2005-SHINING-8,pp.33-38

[3]Microsoft 主催の学生プログラミングコンテスト ”Imagine Cup” ソフトフェア部門にて準優勝

# 付 録A　各ベンチマークにおける実行時間

## A.1　SPEC2000 INT 181.mcf

表 A.1 に 4.3.1 項で評価した実際の実行時間を示す。

表 A.1: SPEC2000 INT 181.mcf の実行時間 [sec]

| シングルスレッドでの実行時間 | | | | | | 289.390 |
|---|---|---|---|---|---|---|
| | $TD_{MIN}$ | | | | | |
| $TD_{MAX}$ | | 1 | 2 | 3 | 4 | 5 |
| | 1 | 281.959 | 279.953 | 282.218 | 283.359 | 288.765 |
| | 2 | — | 281.765 | 280.625 | 280.824 | 288.640 |
| | 3 | — | — | 280.296 | 279.944 | 282.203 |
| | 4 | — | — | — | 284.296 | 279.983 |
| | 5 | — | — | — | — | 284.084 |

## A.2 SPEC2000 INT 300.twolf

表 A.2 に 4.3.2 項で評価した実際の実行時間を示す。

表 A.2: SPEC2000 INT 300.twolf の実行時間 [sec]

| シングルスレッドでの実行時間 | | | | 419.343 |
|---|---|---|---|---|
| | $TD_{MIN}$ | | | |
| $TD_{MAX}$ | | 1 | 2 | 3 |
| | 1 | 536.421 | 523.937 | 522.625 |
| | 2 | — | 528.453 | 518.75 |
| | 3 | — | — | 528.39 |
| 非同期での実行時間 | | | | 513.639 |

## A.3 Olden health

表 A.3 に 4.3.3 項で評価した実際の実行時間を示す。

表 A.3: Olden health の実行時間 [sec]

| シングルスレッドでの実行時間 | | | | | | 193.203 |
|---|---|---|---|---|---|---|
| | $TD_{MIN}$ | | | | | |
| $TD_{MAX}$ | | 1 | 2 | 3 | 4 | 5 |
| | 1 | 230.344 | 229.581 | 228.890 | 228.472 | 228.744 |
| | 2 | — | 229.75 | 229.953 | 230.281 | 230.062 |
| | 3 | — | — | 230.234 | 229.875 | 229.796 |
| | 4 | — | — | — | 230.297 | 230.07 |
| | 5 | — | — | — | — | 230.297 |
| 非同期での実行時間 | | | | | | 190.609 |

# 付 録B　各ベンチマークにおける性能向上率

## B.1　SPEC2000 INT 181.mcf

表 B.1 に 4.3.1 項で評価した実際の性能向上率を示す。

表 B.1: SPEC2000 INT 181.mcf の性能向上率 [%]

| | $TD_{MIN}$ | | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 |
| $TD_{MAX}$ | 1 | 100 | 126.9950209 | 96.514601 | 81.16000538 | 8.410711883 |
| | 2 | — | 102.610685 | 117.9518234 | 115.2738528 | 10.09285426 |
| | 3 | — | — | 101.8082358 | 127.1161351 | 96.71645808 |
| | 4 | — | — | — | 68.55066613 | 126.5913067 |
| | 5 | — | — | — | — | 71.4035796 |

## B.2 SPEC2000 INT 300.twolf

表 B.2 に 4.3.2 項で評価した実際の性能向上率を示す。

表 B.2: SPEC2000 INT 300.twolf の性能向上率 [%]

| | $TD_{MIN}$ | | | |
|---|---|---|---|---|
| | | 1 | 2 | 3 |
| $TD_{MAX}$ | 1 | 100 | 109.3306224 | 110.442204 |
| | 2 | — | 105.7088469 | 113.8965753 |
| | 3 | — | — | 105.7573089 |

## B.3 Olden health

表 B.3 に 4.3.3 項で評価した実際の性能向上率を示す。

表 B.3: Olden health の性能向上率 [%]

| | $TD_{MIN}$ | | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 |
| $TD_{MAX}$ | 1 | 100 | 101.7592302 | 103.4173647 | 104.4519437 | 103.7759593 |
| | 2 | — | 101.363238 | 100.8923935 | 100.1425152 | 100.6417155 |
| | 3 | — | — | 100.2491519 | 101.0726924 | 101.2560864 |
| | 4 | — | — | — | 100.106275 | 100.6233755 |
| | 5 | — | — | — | — | 100.106275 |